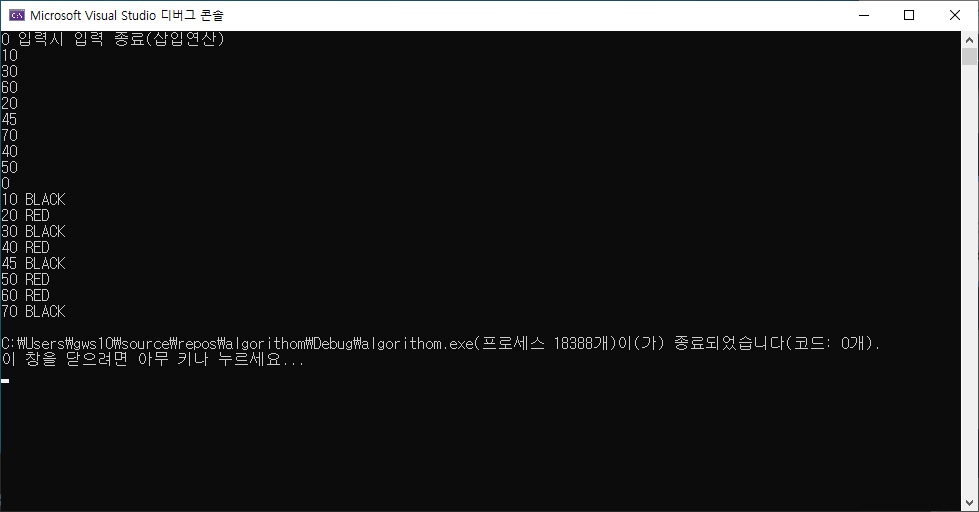
**레드 블랙 트리**



#include <iostream>

using namespace std;

enum Color { RED, BLACK, DOUBLE\_BLACK };

struct Node

{

int data;

int color;

Node\* left, \* right, \* parent;

explicit Node(int);

};

class RBTree

{

private:

Node\* root;

protected:

void rotateLeft(Node\*&);

void rotateRight(Node\*&);

void fixInsertRBTree(Node\*&);

void fixDeleteRBTree(Node\*&);

void inorderBST(Node\*&);

void preorderBST(Node\*&);

int getColor(Node\*&);

void setColor(Node\*&, int);

Node\* minValueNode(Node\*&);

Node\* maxValueNode(Node\*&);

Node\* insertBST(Node\*&, Node\*&);

Node\* deleteBST(Node\*&, int);

int getBlackHeight(Node\*);

public:

RBTree();

void insertValue(int);

void deleteValue(int);

void merge(RBTree);

void inorder();

void preorder();

};

Node::Node(int data) {

this->data = data;

color = RED;

left = right = parent = nullptr;

}

RBTree::RBTree() {

root = nullptr;

}

int RBTree::getColor(Node\*& node) {

if (node == nullptr)

return BLACK;

return node->color;

}

void RBTree::setColor(Node\*& node, int color) {

if (node == nullptr)

return;

node->color = color;

}

Node\* RBTree::insertBST(Node\*& root, Node\*& ptr) {

if (root == nullptr)

return ptr;

if (ptr->data < root->data) {

root->left = insertBST(root->left, ptr);

root->left->parent = root;

}

else if (ptr->data > root->data) {

root->right = insertBST(root->right, ptr);

root->right->parent = root;

}

return root;

}

void RBTree::insertValue(int n) {

Node\* node = new Node(n);

root = insertBST(root, node);

fixInsertRBTree(node);

}

void RBTree::rotateLeft(Node\*& ptr) {

Node\* right\_child = ptr->right;

ptr->right = right\_child->left;

if (ptr->right != nullptr)

ptr->right->parent = ptr;

right\_child->parent = ptr->parent;

if (ptr->parent == nullptr)

root = right\_child;

else if (ptr == ptr->parent->left)

ptr->parent->left = right\_child;

else

ptr->parent->right = right\_child;

right\_child->left = ptr;

ptr->parent = right\_child;

}

void RBTree::rotateRight(Node\*& ptr) {

Node\* left\_child = ptr->left;

ptr->left = left\_child->right;

if (ptr->left != nullptr)

ptr->left->parent = ptr;

left\_child->parent = ptr->parent;

if (ptr->parent == nullptr)

root = left\_child;

else if (ptr == ptr->parent->left)

ptr->parent->left = left\_child;

else

ptr->parent->right = left\_child;

left\_child->right = ptr;

ptr->parent = left\_child;

}

void RBTree::fixInsertRBTree(Node\*& ptr) {

Node\* parent = nullptr;

Node\* grandparent = nullptr;

while (ptr != root && getColor(ptr) == RED && getColor(ptr->parent) == RED) {

parent = ptr->parent;

grandparent = parent->parent;

if (parent == grandparent->left) {

Node\* uncle = grandparent->right;

if (getColor(uncle) == RED) {

setColor(uncle, BLACK);

setColor(parent, BLACK);

setColor(grandparent, RED);

ptr = grandparent;

}

else {

if (ptr == parent->right) {

rotateLeft(parent);

ptr = parent;

parent = ptr->parent;

}

rotateRight(grandparent);

swap(parent->color, grandparent->color);

ptr = parent;

}

}

else {

Node\* uncle = grandparent->left;

if (getColor(uncle) == RED) {

setColor(uncle, BLACK);

setColor(parent, BLACK);

setColor(grandparent, RED);

ptr = grandparent;

}

else {

if (ptr == parent->left) {

rotateRight(parent);

ptr = parent;

parent = ptr->parent;

}

rotateLeft(grandparent);

swap(parent->color, grandparent->color);

ptr = parent;

}

}

}

setColor(root, BLACK);

}

void RBTree::fixDeleteRBTree(Node\*& node) {

if (node == nullptr)

return;

if (node == root) {

root = nullptr;

return;

}

if (getColor(node) == RED || getColor(node->left) == RED || getColor(node->right) == RED) {

Node\* child = node->left != nullptr ? node->left : node->right;

if (node == node->parent->left) {

node->parent->left = child;

if (child != nullptr)

child->parent = node->parent;

setColor(child, BLACK);

delete (node);

}

else {

node->parent->right = child;

if (child != nullptr)

child->parent = node->parent;

setColor(child, BLACK);

delete (node);

}

}

else {

Node\* sibling = nullptr;

Node\* parent = nullptr;

Node\* ptr = node;

setColor(ptr, DOUBLE\_BLACK);

while (ptr != root && getColor(ptr) == DOUBLE\_BLACK) {

parent = ptr->parent;

if (ptr == parent->left) {

sibling = parent->right;

if (getColor(sibling) == RED) {

setColor(sibling, BLACK);

setColor(parent, RED);

rotateLeft(parent);

}

else {

if (getColor(sibling->left) == BLACK && getColor(sibling->right) == BLACK) {

setColor(sibling, RED);

if (getColor(parent) == RED)

setColor(parent, BLACK);

else

setColor(parent, DOUBLE\_BLACK);

ptr = parent;

}

else {

if (getColor(sibling->right) == BLACK) {

setColor(sibling->left, BLACK);

setColor(sibling, RED);

rotateRight(sibling);

sibling = parent->right;

}

setColor(sibling, parent->color);

setColor(parent, BLACK);

setColor(sibling->right, BLACK);

rotateLeft(parent);

break;

}

}

}

else {

sibling = parent->left;

if (getColor(sibling) == RED) {

setColor(sibling, BLACK);

setColor(parent, RED);

rotateRight(parent);

}

else {

if (getColor(sibling->left) == BLACK && getColor(sibling->right) == BLACK) {

setColor(sibling, RED);

if (getColor(parent) == RED)

setColor(parent, BLACK);

else

setColor(parent, DOUBLE\_BLACK);

ptr = parent;

}

else {

if (getColor(sibling->left) == BLACK) {

setColor(sibling->right, BLACK);

setColor(sibling, RED);

rotateLeft(sibling);

sibling = parent->left;

}

setColor(sibling, parent->color);

setColor(parent, BLACK);

setColor(sibling->left, BLACK);

rotateRight(parent);

break;

}

}

}

}

if (node == node->parent->left)

node->parent->left = nullptr;

else

node->parent->right = nullptr;

delete(node);

setColor(root, BLACK);

}

}

Node\* RBTree::deleteBST(Node\*& root, int data) {

if (root == nullptr)

return root;

if (data < root->data)

return deleteBST(root->left, data);

if (data > root->data)

return deleteBST(root->right, data);

if (root->left == nullptr || root->right == nullptr)

return root;

Node\* temp = minValueNode(root->right);

root->data = temp->data;

return deleteBST(root->right, temp->data);

}

void RBTree::deleteValue(int data) {

Node\* node = deleteBST(root, data);

fixDeleteRBTree(node);

}

void RBTree::inorderBST(Node\*& ptr) {

if (ptr == nullptr)

return;

inorderBST(ptr->left);

if(ptr->color == 0) cout << ptr->data << " " << "RED" << endl;

if (ptr->color == 1) cout << ptr->data << " " << "BLACK" << endl;

inorderBST(ptr->right);

}

void RBTree::inorder() {

inorderBST(root);

}

void RBTree::preorderBST(Node\*& ptr) {

if (ptr == nullptr)

return;

if (ptr->color == 0) cout << ptr->data << " " << "RED" << endl;

if (ptr->color == 1) cout << ptr->data << " " << "BLACK" << endl;

preorderBST(ptr->left);

preorderBST(ptr->right);

}

void RBTree::preorder() {

preorderBST(root);

cout << "-------" << endl;

}

Node\* RBTree::minValueNode(Node\*& node) {

Node\* ptr = node;

while (ptr->left != nullptr)

ptr = ptr->left;

return ptr;

}

Node\* RBTree::maxValueNode(Node\*& node) {

Node\* ptr = node;

while (ptr->right != nullptr)

ptr = ptr->right;

return ptr;

}

int RBTree::getBlackHeight(Node\* node) {

int blackheight = 0;

while (node != nullptr) {

if (getColor(node) == BLACK)

blackheight++;

node = node->left;

}

return blackheight;

}

void RBTree::merge(RBTree rbTree2) {

int temp;

Node\* c;

Node\* temp\_ptr = NULL;

Node\* root1 = root;

Node\* root2 = rbTree2.root;

int initialblackheight1 = getBlackHeight(root1);

int initialblackheight2 = getBlackHeight(root2);

if (initialblackheight1 > initialblackheight2) {

c = maxValueNode(root1);

temp = c->data;

deleteValue(c->data);

root1 = root;

}

else if (initialblackheight2 > initialblackheight1) {

c = minValueNode(root2);

temp = c->data;

rbTree2.deleteValue(c->data);

root2 = rbTree2.root;

}

else {

c = minValueNode(root2);

temp = c->data;

rbTree2.deleteValue(c->data);

root2 = rbTree2.root;

if (initialblackheight1 != getBlackHeight(root2)) {

rbTree2.insertValue(c->data);

root2 = rbTree2.root;

c = maxValueNode(root1);

temp = c->data;

deleteValue(c->data);

root1 = root;

}

}

setColor(c, RED);

int finalblackheight1 = getBlackHeight(root1);

int finalblackheight2 = getBlackHeight(root2);

if (finalblackheight1 == finalblackheight2) {

c->left = root1;

root1->parent = c;

c->right = root2;

root2->parent = c;

setColor(c, BLACK);

c->data = temp;

root = c;

}

else if (finalblackheight2 > finalblackheight1) {

Node\* ptr = root2;

while (finalblackheight1 != getBlackHeight(ptr)) {

temp\_ptr = ptr;

ptr = ptr->left;

}

Node\* ptr\_parent;

if (ptr == nullptr)

ptr\_parent = temp\_ptr;

else

ptr\_parent = ptr->parent;

c->left = root1;

if (root1 != nullptr)

root1->parent = c;

c->right = ptr;

if (ptr != nullptr)

ptr->parent = c;

ptr\_parent->left = c;

c->parent = ptr\_parent;

if (getColor(ptr\_parent) == RED) {

fixInsertRBTree(c);

}

else if (getColor(ptr) == RED) {

fixInsertRBTree(ptr);

}

c->data = temp;

root = root2;

}

else {

Node\* ptr = root1;

while (finalblackheight2 != getBlackHeight(ptr)) {

ptr = ptr->right;

}

Node\* ptr\_parent = ptr->parent;

c->right = root2;

root2->parent = c;

c->left = ptr;

ptr->parent = c;

ptr\_parent->right = c;

c->parent = ptr\_parent;

if (getColor(ptr\_parent) == RED) {

fixInsertRBTree(c);

}

else if (getColor(ptr) == RED) {

fixInsertRBTree(ptr);

}

c->data = temp;

root = root1;

}

return;

}

int main() {

int data;

RBTree rbTree;

cout << "0 입력시 입력 종료(삽입연산)" << endl;

cin >> data;

while (data != 0)

{

rbTree.insertValue(data);

cin >> data;

}

rbTree.inorder();

return 0;

}